Lab 04: Testing - TDD

***Note:* Students must demonstrate their lab in the following week of November 2nd, 2020 to get the grade. This lab is worth 20marks. A Missed demonstration will result in 25% grade deduction. Late submission and late demonstration to the lab is welcomed but with a loss of marks. That is, you do not earn full marks for late submission. 10% penalty for every day late up to 50%. Work will not be accepted after 10 days.**
You must talk to your lab professor in case of any unprecedented situation which may result in late submission.
You are required to submit the properly commented (as per assignment submission standards doc on BrightSpace) text file of your code only.

In this lab, you are using the following data structures:

```
typedef int data_t;

typedef struct set {
    data_t *array;
    size_t capacity;
    size_t size;
} set_t;

typedef data_t* set_i_t;
```

Part A asked you to implement the following functions:

```
int set_init( set_t *set, int capacity );
void set_clear( set_t *set);
int set_size( set_t set );
int set_capacity( set_t set );
void set_free( set_t *set );
int set_find( set_t *set, data_t value);
int set_contains( set_t *set, data_t value);
void set_add( set_t *set, data_t value );
void set_remove( set_t *set, data_t value );
data_t set_pop( set_t *set );
```

Manual testing

In this lab, you will alternate between improving your tests and improving your code. This resembles a formal software development process called ***test-driven development*** **(TDD)**. The idea behind TDD is that you should write tests for a program **before** you write the program itself. This requires you to think critically about the expected behaviour of your program even before you begin to code it.

After you write some initial tests, you then write the minimal amount of code necessary to pass the test. After you have fixed any problems and all the current tests are passing, you write more tests and then more code. Gradually, you will build up a large list of tests that help you verify that new changes do not break older features. This technique is also sometimes called the **"test a little, code a little"** strategy.

In this lab, you are going to build a set of tests for part A of your second assignment. You could then use the same strategy to test part B and C.

Ad-hoc Testing

Ad-hoc testing is an informal way of testing. Usually done without planning or documentation. Test is only conducted if an error is found.

This is most likely the type of testing you are conducting at this point. For example,

```
int digits( int a ) {

 int d = 0;

 while( a ) {
       a /= 10;
       d++;
       printf("a = %d\nd = %d\n", a, d );
 }
 return d;
}
```

or in your main, you may have tested as follow:

```
int main( void ) {


 if ( digits( 1000) == 4 )
       printf("It WORKED!!!!!\n");
 else
       printf("NOT WORKING!!!!!\n");

 return EXIT_SUCCESS;
}
```

Notice that in this case, we will need to duplicate the code to test other values and be sure that the function will works.

Automated Testing

It is much more robust to separate a program's tests into a separate module, enabling them to be maintained, compiled, and managed separately. This will allow you to handle test crashes gracefully, and to use the **`main()`** function for actual program code.

In this course you are going to be using the Check framework to do automated testing of some of your labs and assignments.

Installation

To install the check libraries in Ubuntu:

```
sudo apt-get install check
```

Starting Up with testing

Test writing using **`Check`** is very simple.

Create a separate **`.c`** file in your assignment directory (where you are developing your assessements). I am suggesting your test file to be called: **`testProgramName.c (i.e. testSet.c for the 'set' assignment).`**

Since your test is testing your **`set`** assignment, it needs to have access to your data type, function prototypes and to the check framework.

Your **`testSet.c`** must begin with:

```
#include <stdio.h>
#include <stdlib.h>
#include <check.h>
#include "set.h"
```

Your **`Makefile`** already includes the needed directives to compile your test:

```
TEST=test_set
MODS=set_A.o
test: $(TEST)
	./$(TEST)  # This will compile and run your test
TESTCFLAGS=$(CFLAGS) -Wno-gnu-zero-variadic-macro-arguments
TESTLIBS=-lcheck -lm -lpthread -lrt

$(TEST): $(TEST).o $(MODS)
	$(CC) $(TESTLDFLAGS) -o $(TEST) $^ $(LIBS) $(TESTLIBS)

$(TEST).o: $(TEST).c
	$(CC) -c $(TESTCFLAGS) $<
```

To compile your test file:

```
# make test
```

Writing Tests

The basic unit test looks as follows:

```
START_TEST (test_name) {
   /* unit test code */
}
END_TEST
```

The **`START_TEST/END_TEST`** pair are macros that setup basic structures to permit testing. The **`Check`** framework replaces these macros with code that runs the test, handles any crashes, and records the result. Every test is named, and the name is declared as a parameter to the **`START_TEST`** macro.

`Check` provides many convenience functions for performing tests. Here are the most important:

```
/*
 * Evaluate expr;
 * the check passes if the result is true and fails if it is false.
 */
ck_assert(expr);

/*
 * The check passes if the two integers are equal
 * and fails if they are not
 */
ck_assert_int_eq(x, y);

/*
 * The check passes if the two character strings are equal
 *(according to the strcmp function) and fails if they are not.
 */
ck_assert_str_eq(x, y)
```

Let's write a small test to check if your functions set_init( ) is working properly:

```
START_TEST ( init ) {

 /*
  *  Declare a set and initialize with basic values
  */
 set_t set  = { NULL, 0, 0 };

 /*
  *  if the set has been initialized with capacity 10,
  *  then size should be 0 and capacity 10
  */
 set_init(&set, 10);
       set_clear(&set);
       ck_assert(set_capacity(set) == 10);
       ck_assert(set_size(set) == 0);
       set_free(&set);
}
END_TEST
```

Now that you have a test, you can aggregate it into a suit and run them with a suite runner.

```
Suite * test_suite(void) {
    Suite *s;
    TCase *tc_core;
    s = suite_create("Default");
    tc_core = tcase_create("Core");

     tcase_add_test(tc_core, init_clear_size_free);

     suite_add_tcase(s, tc_core);
     return s;
}
int run_testsuite() {
       int fail_nr;
 Suite *s;
 SRunner *sr;
 s = test_suite();
 sr = srunner_create(s);

 srunner_run_all(sr, CK_NORMAL);
 fail_nr = srunner_ntests_failed(sr);
 srunner_free(sr);

 printf("%s\n", fail_nr ? TEST_FAILURE : TEST_SUCCESS );
 return ( !fail_nr ) ? EXIT_SUCCESS : EXIT_FAILURE;
}

int main(int argc, char* argv[]) {
    return run_testsuite();
}
```

where:

```
#define TEST_SUCCESS "SUCCESS: All current tests passed!"
#define TEST_FAILURE "FAILURE: At least one test failed or crashed"
```

The details are outside the scope of this course, in a nutshell, the **`test_suite( )`** function aggregates the test cases you create (add a line of each test you have) and the **`run_testsuite()`** run the test, keeps track of the number of failed assert and prints then end results.

For this lab, you are to write a test to verify the correct functionality of your assignment part A. We will discuss in the lab test plans and how to achieve this.

Solution to one of these 'Array of Struct Pointer' would be:

- By the use of malloc to allocate memory

```
int set_init( set_t *set, int capacity )
{
set->array = (data_t*)malloc(capacity * sizeof(data_t));
if(set->array == NULL)
{
return 1;
}
Else
{
    set->capacity = capacity;
    set->size = 0;
    return 0;
}
}
```

- And a method which free's it

```
void set_free( set_t *set )
{
free(set->array);
set->array = NULL;
set->capacity = set->size = 0;
}
```

- In a separate method I am trying to set all the values in the set to -1 (CLEAR)

```
void set_clear( set_t *set)
{
    int i = 0;
    for (i = 0; i < set->size; i++)
    {
        set->array = CLEAR;
    }
    set->size = 0;
}
```

- Return the Size of the set:

```
int set_size( set_t set )
{
    return sizeof(set->array);
}
```

- Return the capacity:

```
int set_capacity( set_t set )
{
    int capacity = set->capacity;
    return capacity;
}
```

- And then print the set using:

```
void set_print( set_t set )
```

This way try the rest of the functions defined in the Part A above.

Deliverables of this lab: CODE ONLY